Decentralized Event-Based Orchestration

Pieter Hens[1], Monique Snoeck[1], Manu De Backer[1,2,3,4], and Geert Poels[2]

[1] K.U. Leuven, Dept. of Decision Sciences and Information Management, Naamsestraat 69, 3000 Leuven, Belgium
[2] Universiteit Gent, Dept. of Management Information and Operations Management, Tweekerkenstraat 2, 9000 Gent, Belgium
[3] Universiteit Antwerpen, Dept. of Management Information Systems, Prinsstraat 13, 2000 Antwerpen, Belgium
[4] Hogeschool Gent, Dept. of Management and Informatics, Kortrijksesteenweg 14, 9000 Gent, Belgium

Summary. Today, in the state of the art process engine solutions, process models are executed by a central orchestrator (i.e. one per process). There are however a lot of drawbacks in using a central coordinator, including a single point of failure and performance degradation. Decentralization algorithms that distribute the workload of the central orchestrator exist, but they still suffer from a tight coupling and therefore decreased scalability. In this paper, we aim to investigate the benefits of using an event driven architecture to support the communication in a decentralized orchestration. This accomplishes space and time decoupling of the process coordinators and hereby creates autonomous fine grained self-serving process engines. Benefits include an increased scalability and availability of the global process flow.

Keywords: Dynamic workflows, Orchestration, Decentralization.

1 Introduction

In the last couple of years, process modeling got a lot of attention from researchers and practitioners. Especially with the arrival of service oriented computing, process modeling became even more important. Starting from atomic services, new aggregate services can be build by combining the atomic services and describing an execution flow between the different entities. This way composite services are created, which can again be used in other compositions. When these compositions are described with a specific executable language (e.g. BPEL4WS [1]), automated enactment using a process engine can be accomplished. The description of the process flow can be interpreted by a process engine, which coordinates and triggers the described work.

In the current situation, the execution of one process or one composite service is typically coordinated by one central entity (Fig. 1a, coordinator C0). This central coordinator gets a request from a client and starts the execution of the workflow described in the composite service (Fig. 1a, tasks T1, T2 and T3). The coordinator chooses between paths in the workflow that need to be followed, does

M. zur Muehlen and J. Su (Eds.): BPM 2010 Workshops, LNBIP 66, pp. 695–706, 2011.

data manipulation and invokes the necessary (atomic or composite) services. The central coordinator contains all the logic necessary to execute a complex service or process. Note that the coordinator only *routes work*, it doesn't perform any actual work itself. The actual work is executed by the triggered services (Fig. 1a, services S1, S2 and S3). This is called CENTRALIZED ORCHESTRATION [2].

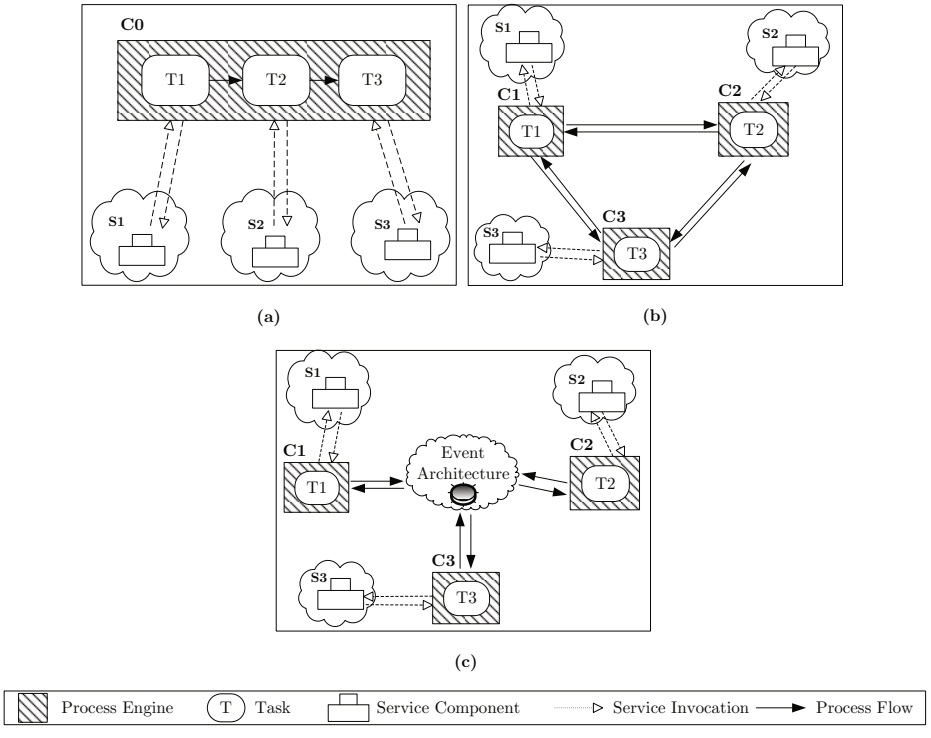


Fig. 1. Centralized, Decentralized and Event-Based Orchestration

The use of a central coordinator per process struggles with a few problems in today's highly decentralized world, and contradicts some of the key aspects of service oriented computing. The use of service oriented computing promises loose coupling (separation of interface and functionality) [3], which increases distributable functionality and reusability [4]. Using a central execution engine or coordinator for a composite service creates a bottleneck and single point of failure. The services (work items) are distributed and decentralized, but the decision logic and coordination is still located at one point. A central coordinator also decreases scalability. For a simple change in the process flow, the entire process description needs to be renewed. Another drawback of a central coordinator is the unnecessary network traffic and performance degradation it creates [5]. This problem of centralized process coordination is also recognized by other researchers [6,7,8].

To overcome this bottleneck, solutions are given to decentralize the coordination work. For example, Nanda et al. [9] define an algorithm to transform a

single BPEL process into multiple (smaller) processes using program dependency graphs. This results in separated process engines, which remove the need for a central coordinator and decentralizes the workflow logic (Fig. 1b, process engines C1, C2 and C3). Advantages of this approach include a significant decrease in network traffic, improved concurrency and availability [5]. This decentralization of the workflow of one composite service is termed DECENTRALIZED ORCHESTRATION.

There are however still a few problems left unsolved in the afore mentioned decentralized orchestration. The execution engines are still mutually tightly coupled, which decreases scalability of the process flow. Each process engine has a direct link to the other process engines that make up the entire composite service. The start of one execution engine in the composite service relies on its invocation by another execution engine. The engines by itself are not autonomous and have to rely on decisions made by others as the logic of the next step in the process is located with the caller, and not with the callee.

In this paper we propose the use of an event based architecture within the decentralized orchestration of a composite service, which we'll term DECENTRALIZED EVENT-BASED ORCHESTRATION (see Fig. 1c). This will create autonomous process engines, capable of assessing their environment and deciding on their own when to invoke their respective service(s) (which is a useful property in process management [10]). An event based communication paradigm also creates a highly loose coupled infrastructure, which makes changes to the process flow relatively easy ('plug and play' of process engines).

The use of an event-driven architecture (EDA) in combination with the service oriented architecture is already thoroughly discussed in research and practice [11,12,13,14]. The difference with these proposals and our work, is the place where the event based communication paradigm is used. In current literature about SOA and EDA, event communication is used for the invocation of services (work items) by the central coordinator (dashed, open arrowhead arrows in Fig. 1). In our approach, we look at the use of an event based architecture *within* the decentralized orchestration of *one* composite service. These are the calls from coordinator to coordinator (full arrowhead arrows in Fig. 1). They correspond with the *control flow* described in one workflow model (sequence, choice, loop).

In the next section we clarify our viewpoint on orchestration and process based composition, and give some definitions of terms used in this paper. In the following two sections we answer the most viable questions for our research: why do we need to decentralize the process flow execution (Sect. 3 and 4) and why is an event based communication a feasible strategy to do so (Sect. 5). In the remaining sections we describe how this can work in practice (Sect. 6) and set a few key points for further research (Sect. 7).

2 Viewpoint

In order to explain the ideas in this paper, we give an overview of our viewpoint on service oriented computing and its link to process composition.

Our goal is to split up one process from one organization in order to distribute the execution load of that process. We look at coordination from a one-process point of view. This enables the abstraction of the interplay between different processes from different entities (organizations, departments, customer-supplier, ...). We are only interested in the control flow (sequence, choice, loop) within one process execution. We also assume that a global process view is at hand. The process modelers have defined a process model that is to be deployed in the company. Decentralization of the model takes place in the deployment stage.

A second viewpoint we take is a strict separation between coordination or process logic and functionality. With coordination logic we mean the *description* of the work that needs to be done. This description can be interpreted by a process engine, which makes sure that all the work that is written down in the description is carried out. The process engine is thus a coordinator, it doesn't perform any work itself. The actual work is executed by services that are triggered by the coordinator. These services contain the functionality or actual work packets of the process description (see Fig. 2). Note that the service that is triggered by the coordinator can be a composite service, which itself contains a process description and thus is a coordinator. Because we look at the problem from a one process point of view, calls to atomic or composite services are treated the same way. They both belong to the functionality side and get triggered by the coordinator. How the actual work is eventually handled isn't of interest to the coordinator of this one process (it just wants the work done). Like it is said in the introduction, we look at the decentralization of the process logic (top part of Fig. 2), not at the invocations of services or other process compositions (bottom part of Fig. 2).

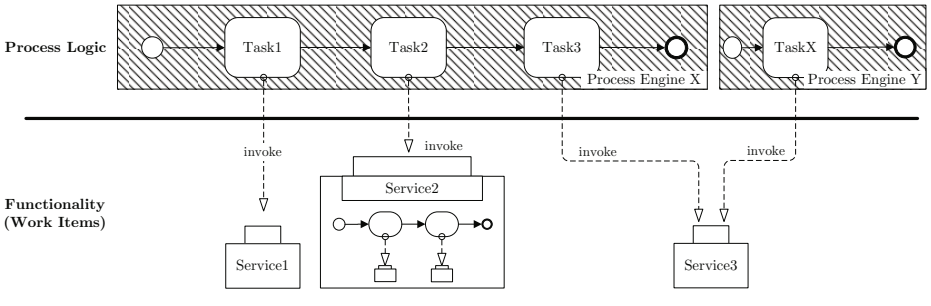


Fig. 2. Separation of the process logic and functionality

This separation of process logic and actual work packets creates a loose coupling between the two. Because the logic on when something has to happen and the work packet itself is separated, the work packets become highly reusable. Other process flows can also trigger the same functions to create new composite services (see Fig. 2, process engine Y). With the help of standard interfaces like WSDL [15] and interaction protocols like SOAP [16], the services and the process logic are not location bound. The services can position themselves anywhere

in the IT infrastructure, the same goes for the process engine(s). This creates a highly distributable process architecture. One thing that is still left as a single monolithic entity is the process description. This paper further investigates the possibilities to distribute this process logic.

2.1 Terminology

To exclude ambiguity, we define some terminology used in this paper.

Event architecture. An event architecture is an architecture that supports the event communication paradigm by implementing a publish/subscribe interaction scheme [17,18]. Events are instantaneous happenings. When an event happens, a notification of its (past) occurrence is routed to interested parties. This routing is done by an event service that also keeps track of which entity is interested in which event and which entity is able to publish which event.

Locus of Control. The locus of control is the place that holds the decision logic of the next step in the process. In a centralized orchestration, there is only one locus of control and it lies with the coordinator. The coordinator knows what and when something has to happen. When decentralizing the process logic, there isn't a single locus of control, but multiple loci of control. At anytime in the process flow the decision logic of the next step in the process can be located at one of the distributed process engines. There can be as much loci of control as there are process engines for a given process.

Orchestration. We adopt the interpretation of orchestration as described by Barros et al. [2]. Orchestration is the execution of the internal actions of one composite service. The execution engine reads and interprets the predefined process flow and invokes the services that have to perform some kind of work.

3 Limitations of Centralized Orchestration

When using a centralized orchestration, two groups of limitations can be identified. First there are technical limitations, like a performance bottleneck and unnecessary data traffic. Second, there could be managerial reasons to not use a centralized orchestration. Technical limitations of a centralized process execution are:

Single point of failure. If the central coordinator fails (either by hardware or software), all process instances will fail. The services capable of performing the work items described in the process flow may still be available (they are distributed), but because of the failure of the coordinator, these services aren't triggered, and therefore aren't executed anymore (see Fig. 3a). Even though there is a high decentralization and distribution of the services described in the process flow, the availability of the entire composite service is still dependent on one single entity: the central coordinator.

Unnecessary network traffic. All (data) traffic runs through the central coordinator. For example, when data generated by one service is important

for another service, this data will be routed through the central coordinator (from the first service to the second service), even if this data is of no importance to the coordinator. This creates a lot of unnecessary network traffic from and to the central coordinator (see Fig. 3b).

Performance bottleneck. Real life processes can build up in scale and complexity and for one process description, multiple process instances can be created (see Fig. 3c). In process intensive organizations, these process instances can run up very quickly. When all the instances are coordinated at one point in the IT infrastructure, performance throughput decreases significantly.

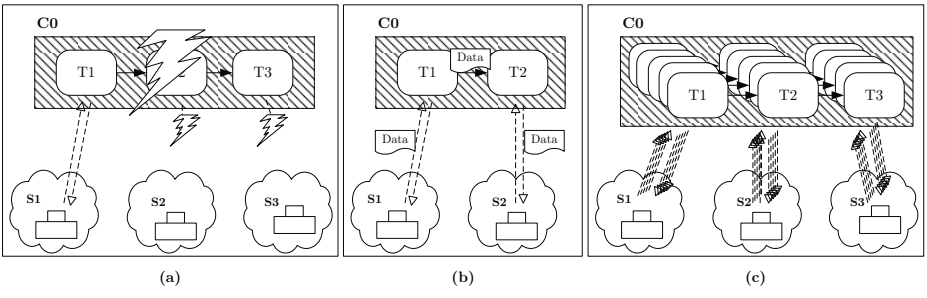


Fig. 3. Limitations of a centralized orchestration: Single point of failure, Unnecessary network traffic and a Performance bottleneck

There are also managerial reasons for not having a central process coordinator. These include security, privacy, visibility, etc. In extended enterprises it is viable that no single organization has control over the entire process (enforced through policies), or that no central entity is allowed to view the entire process, which makes encapsulation necessary.

4 Decentralized Orchestration

The drawbacks of a centralized orchestration have already been recognized by many researchers and practitioners [5,7,8,9,19,20]. To overcome these problems, proposals have been made to decentralize the process execution. These proposals are algorithms which take as input a global process description and give as result a divided process flow. Differences in the algorithms are the way in which the algorithm works, depending on the eventual goal of the division. Nanda et al. [9] use program dependency graphs, a tool borrowed from compiler optimization, to split up the process flow. Their goal is to reduce the network traffic involved. For the same reasons, Fdhila et al. [20] decentralize the process flow using dependency tables and Muth et al. [7] perform decentralization using state and activity charts. Chen et al. [19] use a different division metric, namely business policies, which accomplishes encapsulation of parts of the process flow.

The eventual result is however always the same. After dividing the original process flow, the outcome is a set of control flows, each of which can be interpreted

by a different process engine (see Fig. 4). The coupling between these flows is always tight. The locus of control of one process engine is located at another engine. The start of a process flow is dependent on the invocation done by another process flow. Each process engine thus knows what the next step in the global process flow should be ("I am done, now I'll request the start of engine X"). This is the source of the tight coupled communication.

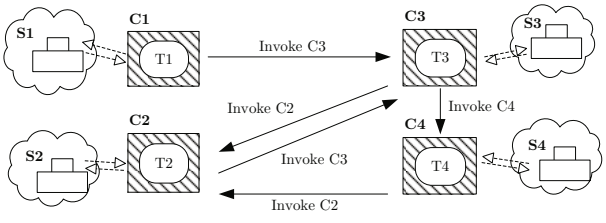


Fig. 4. Decentralized Orchestration

This decentralization thus solves the problems associated with centralized orchestration, but not to a full extend. Because of tight coupling, scalability is still a problem. The availability of the global process flow can also still be optimized. This is why we look at the use of an event based communication paradigm.

5 Event-Based Orchestration

We propose to take the decentralization one step further and extend the decentralization of the coordination work with an event-based communication paradigm. This will create highly loose coupled, autonomous process engines. The use of an event-based architecture to accomplish loose coupling has already been thoroughly studied in process modeling and other domains [12,17,18]. The novelty of this proposal is to use the event-based architecture to decouple the internal process flow. Starting from a global process flow, it is the next step in the decentralization algorithms (from central to decentralized to event based process execution).

Using an event architecture as communication scheme between the decentralized process engines is shown in Fig. 5. Each decentralized coordinator listens to its environment and reacts accordingly. A single process engine doesn't invoke the next step in the process flow anymore, it just publishes a notification of the event that just happened ("job X is done"). This leaves the decision on what the next step in the global process is, completely in the hands of the next step itself.

The advantage of using an event architecture is decoupling between the sender and receiver of a message. We validate the usefulness of using an event architecture in decentralized orchestration by looking at this decoupling. Decoupling in event architectures is defined by Eugster et al. [17], who give three meanings to decoupling accomplished by using an event architecture: space decoupling, time decoupling and synchronization decoupling.

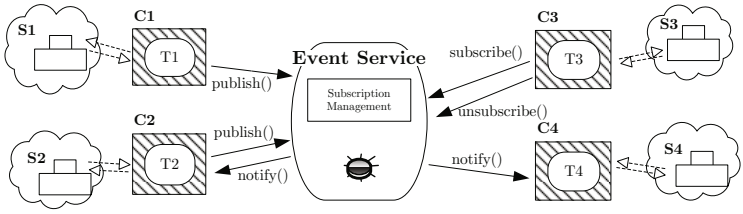


Fig. 5. Decentralized Event-Based Orchestration

Space decoupling. Space decoupling refers to the unawareness of interaction partners. Publishers publish events without knowing who receives them and subscribers consume these events, without knowing who sent them. When using an event-based architecture, space decoupling is the biggest contribution to decentralized orchestration. The locus of control shifts from the sender to the receiver of a message (or event notification). This creates autonomous process engines. The engines themselves decide when they start the execution of the process flow, they don't rely on a decision made by another process engine. Space decoupling also shields the different process engines from each other, which increases scalability. Because a process engine doesn't rely on others, but rather on its own decision on when to start the process, a plug and play architecture becomes feasible. Even when using automated transformations from a global process flow to a decentralized one, a plug and play ability to change the deployed process flow is a definite plus. With a tight coupled decentralization, even small changes in the process flow result in big changes to all the different process engines (which can become cumbersome when they are all physically distributed in far away locations). Because of space decoupling, changing the process flow influences only those process engines which are actually involved in that specific change.

Time decoupling. Interacting parties do not need to be active at the same moment in time. This allows a process engine to be offline, while others continue their regular course of action. Time decoupling, together with a distributed location of the process engines, increases the protection of the global process against a single point of failure. A process flow, implemented with an event communication scheme, doesn't get interrupted when one (or more) process engines in the process flow fail. The still active entities keep working and when the failed engine comes back online, all published events of interest that happened during the failure, will still be delivered. Time decoupling thus guarantees availability of the global process flow.

Synchronization decoupling. Synchronization decoupling refers to the asynchronous send and receive of messages. The sender, nor the receiver gets blocked while sending or receiving events. Asynchronous invocations are already present in current orchestration decentralizations. In terms of synchronization decoupling, an event architecture doesn't add a surplus value versus classical decentralized orchestration.

Space and time decoupling are an added value of using an event-based communication paradigm in a decentralized orchestration. They will increase the scalability and availability of the process flow.

High decoupling of sender and receiver in decentralized orchestration is however not a silver bullet. A few disadvantages of using an event communication paradigm between the partial process flows can be identified:

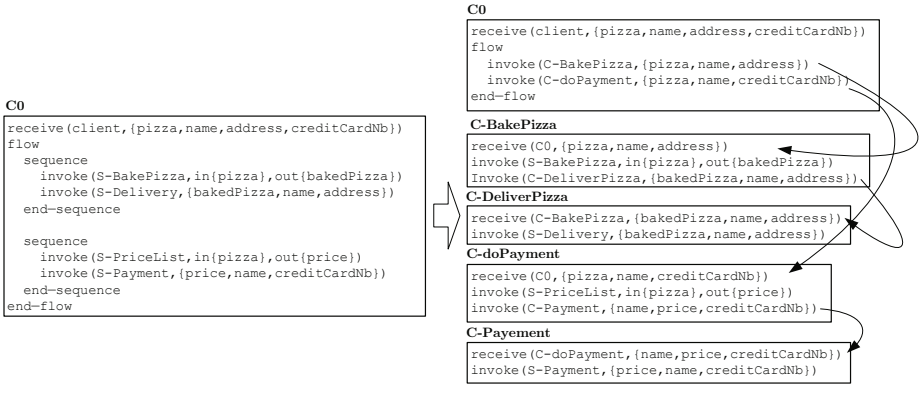
Loss of a global process overview. A disadvantage of using an event architecture is that, at runtime, the global overview of the process flow gets lost. Every process engine works autonomously, without the knowledge of any other part of the process flow. The execution of the process flow thus becomes stateless. Although the overview of the process flow gets lost at runtime, there still exists a process model. Decentralization happens at deployment, where one starts with a global process overview. This overview will thus still be available. In service oriented computing it is also advised to design services in a stateless way [4,21]. Using stateless process engines thus fits in this ideal. Inspection of the process state at runtime can still be done by monitoring the events at runtime and relating them with the global process model.

Loss of coupling process instance and action. Related to the loss of the global process view, is the difficulty of relating events to process instances. If a decentralized process engine subscribes to two events, e.g. `OrderCreated` and `PaymentComplete`, and defines in its process flow that the order should be shipped when it gets notified of these two events, it is imperative that these two events belong to the same process instance (order) before it ships the goods. A way to insure instance-event coupling is to add instance ids to each event (similar to *correlation sets* in BPEL). For every new request from the client, the published events, and any following, get the same instance id. This can be accomplished by the event architecture itself. This way a process engine shall always perform the necessary actions for the correct process instance.

Reduced coordination. Because of space decoupling, an event communication paradigm strongly reduces coordination between interacting partners, which creates a particular challenge for transactions. To accomplish a transaction, knowledge of participating partners is required, which conflicts with the space decoupling. This problem can be overcome by intelligent modularization of the process flow, i.e. keeping tasks in a transaction in the same partial process engine.

6 In Practice - Case Study

When starting from a global process description, any process decentralization algorithm presented in Sect. 4 can be used to first create decentralized process engines. These engines then have to be modified to use the event communication paradigm. To accomplish the resulting publishing and catching of events in a service oriented environment, event architectures like WS-Notification [14], EVE [22] or the more recently proposed BPEL and WSDL extensions for an event driven architecture [23] can be used.



(a) Centralized Orchestration **(b) Decentralized Orchestration**

Fig. 6. From centralized process execution to decentralized execution

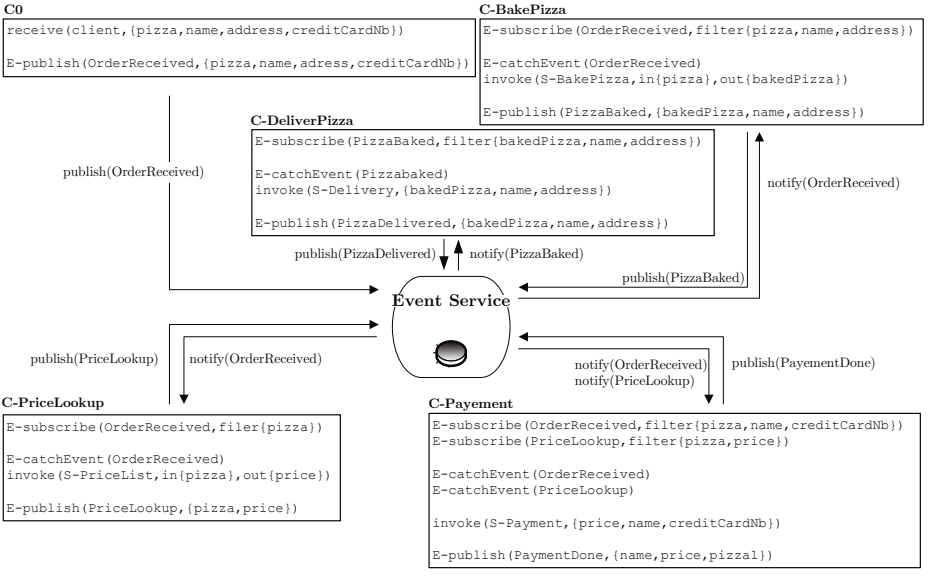


Fig. 7. Decentralized event based execution

We give a simple example to demonstrate the two steps involved in transforming a global process flow to a decentralized event based one, hereby showing the feasibility of this approach. A pizza delivery company accepts orders from clients. If an order is received, the payment of the order and the baking + delivery of the pizza are executed in parallel. Delivery is only started after the baking is complete, and payment is only started after price calculation. This process is shown in Fig. 6a. We used a pseudo-BPEL code, as introduced by [9] to describe the process flow. The first step is to decentralize this process. We did this using

the decentralization algorithm described in [9]. Each resulting process engine calls the next one by means of `invoke` and `receive` operations (see Fig. 6b). The second step is to change this to event communication. For every coordinator to coordinator communication, each `invoke` operation is changed to a publish of the recent happenings: `E-publish(eventName,payload)`, and every `receive` operation is changed to accept notifications: `E-catchEvent(eventName)`. This also means adding a subscription to this notification in the beginning of the process flow: `E-subscribe(eventName,filter)`. Note that invocations from the decentralized process engines to their respective services (functionality) remain unchanged.

Figure 7 shows the resulting pseudo BPEL for the different coordinators. A big difference we see with Fig. 6b is the non-occurrence of references to the other coordinators. In Fig. 6b each coordinator still has a hard-coded link to another one (see the arrows). This is removed when an event driven architecture is used. The global process flow is now executed by loosely coupled, autonomous coordinators.

7 Conclusion and Future Research

In this paper, we promoted and examined the idea of using an event driven architecture to further extend decentralized orchestration. An added value of the use of an event driven communication paradigm is space and time decoupling between the decentralized orchestration engines. This increases the scalability and availability of the global process flow and creates autonomous process engines, which can be deployed at runtime (plug and play) and can be distributed in the global IT infrastructure. With an example we showed the feasibility of this transformation to an event based orchestration.

Further research involves the formalization of these transformation rules from a global process model to a decentralized event based orchestration. We intend to prove the correctness of these transformation rules with process algebra and formally validate the added value (see Sect. 5), by testing on availability (stress testing) and scalability of the decentralized event-based process flow.

References

1. Oasis: Web service business process execution language version 2.0. Oasis Standard
2. Barros, A., Dumas, M., Oaks, P.: Standards for web service choreography and orchestration: Status and perspectives. In: BPM Workshops, pp. 61–74
3. Papazoglou, M.: Extending the service-oriented architecture. Business Integration Journal 7(1), 18–21 (2005)
4. Erl, T.: SOA: Principles of service design. Prentice Hall Press, NJ (2007)
5. Chafle, G., Chandra, S., Mann, V., Nanda, M.: Decentralized orchestration of composite web services. In: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, pp. 134–143 (2004)
6. Benatallah, B., Dumas, M., Sheng, Q.: Facilitating the rapid development and scalable orchestration of composite web services. Distributed and Parallel Databases 17(1), 5–37 (2005)

7. Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A., Weikum, G.: From centralized workflow specification to distributed workflow execution. Journal of Intelligent Information Systems 10(2), 159–184 (1998)
8. Yu, W.: Decentralized Orchestration of BPEL Processes with Execution Consistency. In: Advances in Data and Web Management, pp. 665–670
9. Nanda, M., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. ACM SIGPLAN Notices 39(10), 170–187 (2004)
10. Jennings, N., Norman, T., Faratin, P., OBrien, P., Odgers, B.: Autonomous agents for business process management. Applied Artificial Intelligence 14(2) (2000)
11. Michelson, B.: Event-driven architecture overview. OMG report (2006)
12. Pedrinaci, C., Moran, M., Norton, B.: Towards a Semantic Event-Based Service-Oriented Architecture. In: Workshop: 2nd International Workshop on Semantic Web Enabled Software Engineering, SWESE 2006 (2006)
13. Sriraman, B., Architect, L., Radhakrishnan, R., Architect, E.: Event Driven Architecture Augmenting Service Oriented Architectures. Sun Microsystems (2005)
14. Niblett, P., Graham, S.: Events and service-oriented architecture: the OASIS web services notification specifications. IBM Systems Journal 44(4), 869–886 (2005)
15. Chinnici, R., Gudgin, M., Moreau, J., Weerawarana, S.: Web services description language (WSDL) version 1.2 part 1. W3C Working Draft 11 (2003)
16. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A., Lafon, Y.: SOAP Version 1.2. W3C Working Draft 9 (2001)
17. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) 35(2), 131 (2003)
18. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer-Verlag New York, Inc., Secaucus (2006)
19. Chen, Q., Hsu, M.: Inter-enterprise collaborative business process management. In: International Conference on Data Engineering, p. 0253 (2001)
20. Fdhila, W., Yildiz, U., Godart, C.: A flexible approach for automatic process decentralization using dependency tables. In: ICWS 2009: Proceedings of the 2009 IEEE International Conference on Web Services, pp. 847–855. IEEE Computer Society, Washington, DC, USA (2009)
21. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series). Prentice Hall PTR, NJ (2004)
22. Geppert, A., Tombros, D.: Event-based distributed workflow execution with EVE. In: Proc. of the IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing, pp. 427–442 (1998)
23. Juric, M.B.: Wsdl and bpel extensions for event driven architecture. Information and Software Technology (2010) (in press, accepted manuscript)